Exhibit 7

Exhibit 7

H D T P
S P E C I F I C A T I O N

V e r s i o n 1 . 1 — D R A F T

UNWIRED PLANET, INCORPORATED

390 Bridge Parkway
Redwood Shores, California 94065
USA

Part Number HDTP-SPEC-DOC-101

July 15, 1997

IMPORTANT NOTICE

Preface

The Handheld Device Transport Protocol (HDTP) is a datagram protocol to perform secure, lightweight client/server transactions. This specification defines HDTP version 1.1.

DISCLAIMER: This is a draft specification. The contents herein are subject to change.

Introduction

The World Wide Web provides a robust, flexible, and ubiquitous model for information access. The adoption of the WWW as the preferred means of disseminating and accessing information from desktop PCs and workstations has created a demand for access to the same information from other devices. These devices or “alternate platforms” range from voice- and fax-based user agents to low-cost Network Computers to hand-held devices such as mobile phones and PDAs.

While the web's infrastructure and protocols fully support most of these alternate platforms, many of the protocols are neither practical nor easily implemented on certain devices. In particular, the reliance of HTTP on the TCP/IP protocol stack and the packet, processing and memory overhead of SSL make them poor protocols for many wireless networks. HDTP provides a protocol semantically equivalent to HTTPS (HTTP and SSL), allowing hand-held devices on low-performance, high-cost networks to function as full-fledged web clients.

Hand-held Devices

While there are many types, styles, and classes of hand-held devices, this specification is useful for a significantly larger class of devices with similar physical characteristics. Those characteristics include:

- limited bandwidth
- limited resources such as memory, processing power, permanent storage

Network bandwidth is usually low due to limitations of the network technology or simple economics. The same goes for other resources: memory, processing power, even battery life. While some hand-held devices do have large amounts of memory or

processing power, these devices are the exception. Mass market and consumer-targeted devices will continue to have these constraints for many years.

This specification uses the term “hand-held,” recognizing that the term is not inclusive of all devices which would benefit from HDTP.

HDTP Origins

HDTP originated with the Unwired Planet UP.Link Platform™ as the protocol between an ultra-lightweight web browser, the UP.Browser™, and a proxy server, the UP.Link Gateway™. As such, the protocol is tuned for browser-proxy communication, but that should not prevent the protocol from being used for more general applications.

The first incarnation of HDTP was UGP, the UP.Link Gateway Protocol. The second incarnation of the protocol was SUGP, the Secure UP.Link Gateway Protocol. In SUGP, security was added, the notification model was restructured, and the protocol was further generalized. HDTP 1.1 is derived from SUGP 1.0, with more generalization, removal of quirks, and a tighter coupling to HTTP.

1 Overview

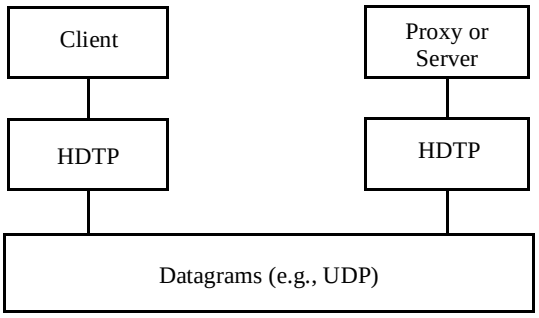
HDTP (Handheld Device Transport Protocol) is an application- and session-level protocol for remote operations between a client and proxy or server. It provides the same basic functionality of HTTPS (i.e., HTTP over SSL), namely:

- Extensible request methods
- Typing and negotiation of data representation
- Message privacy, integrity and authentication

Moreover, HDTP has been optimized for small clients (such as handheld devices) and low-performance networks in the following ways:

- The protocol is datagram-based rather than stream-based allowing for a simpler underlying protocol stack.
- Header names and other well-known values are encoded wherever possible.
- Session contexts reduce transmission of redundant data.
- Asynchronous notifications support data push without the polling of traditional push systems.
- Security and cipher algorithms used can be implemented on small devices.

FIGURE 1-1. HDTP Protocol Stacks



Features

The following features are provided by HDTP.

HTTP Semantics

HDTP provides semantics similar to HTTP. Server requests and responses may include both headers (meta-information) and content. The following HTTP methods are supported:

- OPTIONS
- GET
- HEAD
- POST
- PUT
- DELETE

Sessions

Requests and replies are performed within the context of a session. The session context typically contains the following information:

- Client and Server SessionId
- Session Key
- Cipher
- Protocol Version
- Current RequestId
- DeviceId

A special meta-session (session 0) is reserved for the creation of sessions. Session 0 is unencrypted and unauthenticated, and can accept any protocol version.

Each session has associated server and client session identifiers. Requests carry a four byte server session id so that the server can support simultaneous sessions with a large number of clients. Replies carry a one byte client session id so that the client can maintain multiple simultaneous sessions as well. Only one byte is used for replies to reduce packet size and because it is anticipated that clients will have no more than 255 active sessions.

As part of the session creation process, protocol version, cipher algorithm, and session key are negotiated between client and server. Also, constant request headers, and other parameters are exchanged between the client and server.

Session creation is designed to locate the complex calculations on the server. For example, the random session key is generated by the server and communicated to the client.

Security

HDTP provides a number of security features.

Authentication, Privacy, and Integrity

HDTP provides for authentication, privacy and integrity. Sessions are authenticated. The session authentication between a client and server is conducted by using shared secret key and a nonce challenge. All protocol data units (PDUs) sent in a session are encrypted. Message integrity is enforced by performing a checksum of the entire PDU and encrypting the checksum together with the PDU.

HDTP provides means to support different cipher algorithms. As part of a session creation request, the cipher algorithm is negotiated between client and server.

Note that key management, namely the provisioning of the shared secret key, is outside the scope of this specification.

Counteracting Playback Attacks

Playback is a very common security attack. The playback attack scenario is that an attacker makes copies of messages sent from client to server, and plays them back at a later date, trying to cause the server to respond to the forged requests. HDTP provides means to counteract playback attacks.

For regular transactions, a request identifier (RequestId) is used to detect playback requests. The RequestId is encrypted in the protocol PDU. Every request and reply must carry a RequestId. During the lifetime of a session, any RequestId in requests and replies must equal to the current RequestId in both server and client.

For the session creation process, both client and server challenge each other by using a nonce as a challenge and nonce + 1 as a response to the challenge to avoid playback of session creation requests.

PDU Piggybacking

Where possible, HDTP allows PDU piggybacking to reduce the number of packets sent over the network. For example, the session creation process requires the client to send at least two messages (*Session Request* and *Session Complete*) to the server. The second message (*Session Complete*) can be combined with the regular service request that triggered the session creation in the first place.

Versioning

HDTP provides a version negotiation mechanism, so that the protocol may evolve while remaining backwards or forwards compatible in the future. The protocol version is negotiated during session creation.

2 Message Formats

This section describes the data formats used to exchange HDTP messages between client and server.

Data Formats

The following data types are used in the message format definitions.

TABLE 2-1. Format Definition Data Types

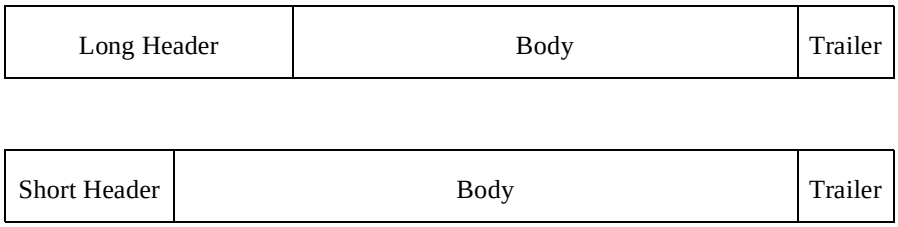
Data Type	*Definition*
bit	1 bit of data
byte	8 bits of opaque data
char	8 bit signed integer
u_char	8 bit unsigned integer
short	16 bit signed integer
u_short	16 bit unsigned integer
int	32 bit signed integer
u_int	32 bit unsigned integer

Network byte order for multi-byte integer values is “big-endian”. In other words, the most significant byte is transmitted on the network first followed subsequently by the less significant bytes.

Network bit ordering for bit fields within a byte is “big-endian”. In other words, bit fields described first are placed in the most significant bits of the byte.

Message Structure

FIGURE 2-2. Message Formats



Every message has a *header*, a *body* and a *trailer*. The header identifies the session the message is destined for. The body contains one or more *Protocol Data Units* (PDUs). Each PDU performs a particular function in the protocol and contains type-specific content. The trailer contains any encryption-related data required by the session's cipher algorithm, such as checksums, padding bytes and initialization vectors.

There are two kinds of headers: *long* and *short*. The long header has 4 bytes and is used for messages sent from the client to the server. The short has 1 byte and is used for messages sent from the server to the client.

The maximum length of the entire message is determined by the underlying datagram transport.

Long Header

The long header is used in messages destined for the HDTP server.

TABLE 2-3. Long Header Fields

Name	*Type*	*Purpose*
SessionId	u_int	Server session identifier

Server SessionId values are specified in Table A-1 in Appendix A.

Session 0 is a special session used to create user sessions and perform other meta-functions. Session 0 is inherently unencrypted and unauthenticated. This means:

- Session 0 will respond only to PDUs that include authentication information.
- Any encryption required in a PDU destined for session 0 is defined in the PDU format.
- The trailer is of length zero.

Sessions 0x00000001–0x0000000F are reserved for future use. Sessions 0x00000010–0xFFFFFFFF are used for user sessions.

Short Header

The short header is used in messages destined for the HDTP client.

TABLE 2-4. Short Header Fields

Name	*Type*	*Purpose*
SessionId	u_char	Client session identifier

Client SessionId values are specified in Table A-2 in Appendix A.

Session 0 is a special session used for certain meta-functions and error messages. Session 0 is inherently unencrypted and unauthenticated. This means:

- Session 0 will only accept PDUs that include some form of authentication or validation information.
- Any encryption required in a PDU destined for session 0 is defined in the PDU format.
- The trailer is of length zero.

Sessions 0x01–0x02 are reserved for SUGP backwards compatibility. Sessions 0x03–0x0F are reserved. Sessions 0x10–0xFF are user sessions.

Protocol Data Units (PDUs)

The body of an HDTP message contains one or more *protocol data units* (PDUs). Each PDU serves a particular function in the protocol and contains type-specific information.

PDU Categories

PDUs fall into one of four categories: requests, replies, acknowledgments or other. The basic protocol is a three-way handshake, where the client sends a request, the server a reply, and the client an acknowledgment. This is described further in "Protocol Operations".

Requests

The following PDUs are considered requests:

- SessionRequest
- Get
- GetNotification
- Post
- Options
- Head
- Put
- Delete

Replies

The following PDUs are considered replies:

- SessionReply
- Reply
- Error
- Redirect

Acknowledgments

The following PDUs are considered acknowledgments:

- SessionComplete
- Cancel
- Ack

Other

The following PDUs perform other functions:

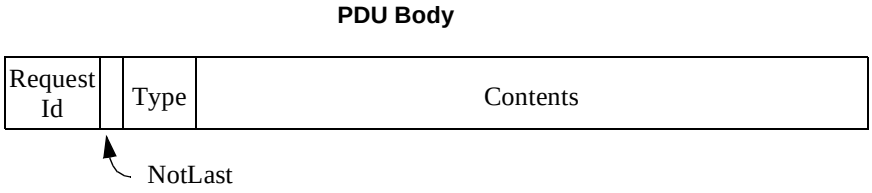
- HoldOn
- Signal

PDU Common Fields

This section describes fields that are common across all or many PDUs.

PDU Header

FIGURE 2-5. PDU Header Format



Every PDU starts with a request identifier (RequestId), a NotLast bit, and type identifier (Type).

TABLE 2-6. PDU Header Fields

Name	*Type*	*Purpose*
RequestId	u_char	Request identifier
NotLast	1 bit	Not the last PDU in the message
Type	7 bits	Type and function of this PDU

The RequestId field is used to identify duplicate messages and to counteract playback attacks. All request, reply, and error PDUs associated with a single transaction share the same RequestId.

RequestId's must be monotonically increasing, and may not be repeated. A side-effect of this is that there can be no more 256 transactions during the lifetime of a session. This is an intentional counter to a clear-text attack on the session key.

The NotLast bit indicates that this PDU is not the last PDU in the message. In other words, another PDU immediately follows this PDU.

The Type field specifies the function of the PDU. The rest of the PDU is type-specific information, referred to as the *contents*. The following sections describe the format of the contents for each PDU type. The type numbers for the various PDUs are summarized in Table A-3 in Appendix A.

In the interest of brevity, the PDU header has been omitted from the description of each PDU in the sections that follow.

Headers

Many PDU types contain a *headers* field. The headers field contains zero or more key-value pairs of strings. Headers usually contain meta-information about the request or response.

Each key-value pair in the headers is comprised of two null (0x00) terminated strings, the key and the value. For example, if there were a header with the key "Name" and the value "Bob", it would be encoded as follows:

```
'N' 'a' 'm' 'e' 0x00 'B' 'o' 'b' 0x00
```

Well-known and frequently used header keys and values may be encoded as a single character string where the character is between hexadecimal 0x80 and 0xFF. For example the HTTP "Content-type" header key is encoded as decimal 0x9A, and the content-type value "application/x-hdmlc" is encoded as 0x81. Thus, the header "Content-type: application/x-hdmlc" can be encoded as follows:

```
0x9A 0x00 0x81 0x00
```

Note that the following encodings of "Content-type: application/x-hdmlc" are considered equivalent to the previous one:

```
0x9A 0x00 'a' 'p' 'p' 'l' 'i' 'c' 'a' 't' 'i' 'o' 'n' '/' 'x'
'-' 'h' 'd' 'm' 'l' 'c' 0x00

'C' 'o' 'n' 't' 'e' 'n' 't' '-' 'T' 'y' 'p' 'e' 0x00 0x81 0x00

'C' 'o' 'n' 't' 'e' 'n' 't' '-' 'T' 'y' 'p' 'e' 0x00 'a' 'p'
'p' 'l' 'i' 'c' 'a' 't' 'i' 'o' 'n' '/' 'x' '-' 'h' 'd' 'm'
'l' 'c' 0x00
```

Well-known header keys are listed in Table A-5 in Appendix A. Well-known header values for particular header keys are listed in subsequent tables.

Request PDUs

SessionRequest

SessionRequest is a request PDU to initiate the creation of a session.

TABLE 2-7. SessionRequest Contents

Name	*Type*	*Purpose*
Cipher	2 bytes	Cipher algorithm and parameters
Version	u_char	HDTP protocol version
ClientSessionId	u_char	Client identifier for the new session
DeviceIdLen	u_char	Length of DeviceId field
HeadersLen	u_short	Length of Header field
DeviceId	*DeviceIdLen* bytes	Device identifier (max. 255 bytes)
Headers	*HeadersLen* bytes	Session headers (max. 65535 bytes)
ClientNonce	u_short	Client nonce challenge for authentication
EncryptionTrailer	variable	Trailer for nonce encryption

The SessionRequest PDU is unusual in that some of its fields, namely the ClientNonce and EncryptionTrailer are encrypted. The names of the encrypted fields are shown in gray in the above table. Normally, the SessionRequest is sent to session 0 which, by definition, is an unencrypted session. If the SessionRequest is sent to an encrypted user session, the net result will be that the encrypted fields in the PDU will be doubly encrypted.

The Cipher field identifies the cipher algorithm and its cipher-specific parameters used by the device to encrypt the encrypted fields. It also acts as the proposed cipher to be used for the new session. The first byte of the Cipher field contains the cipher algorithm. Encoded values for the cipher algorithm are listed in Table A-4 in Appendix A. The second byte of the Cipher field contains cipher-specific parameters.

The Version field identifies the version of the HDTP protocol. This is used to determine the formats of this and all subsequent PDUs. The version number is encoded as follows: The major number of the version is stored in the high-order 4 bits, and the minor number is stored in the low-order 4 bits.

The ClientSessionId field contains the client session identifier to use when sending messages to the new session on the client.

The DeviceIdLen and HeadersLen fields specify the length of the DeviceId and Headers fields respectively.

The DeviceId field contains an opaque device identifier. The contents of the device identifier are defined by the server.

The Headers field contains headers that apply to the entire session. Depending on the header key, these can be headers that will be automatically attached to subsequent service requests, or they can be session-specific parameters.

The ClientNonce field contains a nonce to be used as an authentication challenge to the server. Playback attacks are prevented by virtue of the fact that a nonce is a non-repeating number. Thus, every SessionRequest is unique in that it will always have a different nonce.

Get

Get is a request PDU analogous to the HTTP GET method. It is sent to get whatever information is associated with a given URL.

TABLE 2-8. Get Fields

Name	*Type*	*Purpose*
HeadersLen	u_short	Length of the Headers field
UrlLen	u_short	Length of the Url field
Headers	*HeadersLen* bytes	Request headers
Url	*UrlLen* bytes	URL to get

The HeadersLen and UrlLen fields specify the length of the Headers and Url fields respectively.

The Headers field contains the headers associated with the request.

The Url field contains the Url to get.

GetNotification

GetNotification is a request PDU sent to retrieve any outstanding data pushes. The GetNotification PDU has no type-specific contents. The server will respond with a Reply PDU containing the pushed data.

GetNotification differs from Get in that it does not request a particular URL. The server knows the URL for the notification, not the client. The reply from the server should include the "Location" header to indicate the URL of the data pushed.

Post

Post is a request PDU analogous to the HTTP POST method. It is used to request that the server accept the entity enclosed in the request as a new subordinate of the resource identified by the URL.

TABLE 2-9. Post Fields

Name	*Type*	*Purpose*
HeadersLen	u_short	Length of Headers field
UrlLen	u_short	Length of Url field
DataLen	u_short	Length of Data field
Headers	*HeadersLen* bytes	Request headers

Name	Type	Purpose
Url	*UrlLen* bytes	URL to post to
Data	*DataLen* bytes	Data to be posted

The HeadersLen, DataLen and UrlLen fields indicate the size of the Headers, Data, and Url fields respectively.

The Headers field contains the headers associated with the request.

The Url field contains the Url to post to.

The Data field contains the data associated with the request.

Options

Options is a request PDU analogous to the HTTP OPTIONS method. It is used as a request for information about the communication options available on the request/response chain identified by the URL. This method allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or initiating a resource retrieval.

TABLE 2-10. Options Fields

Name	Type	Purpose
HeadersLen	u_short	Length of Headers field
UrlLen	u_short	Length of Url field
Headers	*HeadersLen* bytes	Request headers
Url	*UrlLen* bytes	URL for the options to get

The HeadersLen and UrlLen fields indicate the size of the Headers and Url fields respectively.

The Headers field contains the headers associated with the request.

The Url field contains the Url for the options to get.

Head

Head is a request PDU analogous to the HTTP HEAD method. The Head PDU is identical to Get except that the server MUST NOT return a message-body in the response. The meta-information contained in the headers in response to a Head request SHOULD be identical to the information sent in response to a Get request. This method can be used for obtaining meta-information about the entity implied by the request without transferring the entity-body itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification.

TABLE 2-11. Head Fields

Name	Type	Purpose
HeadersLen	u_short	Length of Headers field
UrlLen	u_short	Length of Url field

Name	*Type*	*Purpose*
Headers	*HeadersLen* bytes	Request headers
Url	*UrlLen* bytes	URL for the headers to get

The HeadersLen and UrlLen fields indicate the size of the Headers and Url fields respectively.

The Headers field contains the headers associated with the request.

The Url field contains the Url for the headers to get.

Put

Put is a request PDU analogous to the HTTP PUT method. It is used to request that the enclosed entity be stored under the supplied URL. If the URL refers to an already existing resource, the enclosed entity should be considered as a modified version of the one residing on the origin server. If the URL does not point to an existing resource, and that URL is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URL.

TABLE 2-12. Put Fields

Name	*Type*	*Purpose*
HeadersLen	u_short	Length of Headers field
UrlLen	u_short	Length of Url field
DataLen	u_short	Length of Data field
Headers	*HeadersLen* bytes	Request headers
Url	*UrlLen* bytes	URL to put
Data	*DataLen* bytes	Data to put

The HeadersLen, UrlLen and DataLen fields indicate the size of the Headers, Url, and Data fields respectively.

The Headers field contains the headers associated with the request.

The Url field contains the Url to put.

The Data field contains the data associated with the request.

Delete

Delete is a request PDU analogous to the HTTP DELETE method. It is used to request that the server delete the resource identified by the URL.

TABLE 2-13. Post Fields

Name	*Type*	*Purpose*
HeadersLen	u_short	Length of Headers field
UrlLen	u_short	Length of Url field

Name	Type	Purpose
Headers	*HeadersLen* bytes	Request headers
Url	*UrlLen* bytes	URL to delete

The HeadersLen and UrlLen fields indicate the size of the Headers and Url fields respectively.

The Headers field contains the headers associated with the request.

The Url field contains the Url to delete.

Reply PDUs

SessionReply

SessionReply is a reply PDU in response to the SessionRequest PDU. The SessionReply is sent to the client session identified in the SessionRequest. Thus, the entire PDU should be encrypted.

New client sessions, by default, expect messages to be encrypted using the cipher proposed in the SessionRequest and the shared secret key. The SessionReply contains a SessionKey and proposed cipher to be used for subsequent communication.

TABLE 2-14. SessionReply Fields

Name	Type	Purpose
ServerNonce	u_short	Server nonce challenge for authentication
ClientNoncePlus	u_short	ClientNonce+1 response for authentication
ServerSessionId	u_int	Server session identifier
Cipher	2 bytes	Cipher algorithm and parameters
SessionKeyLen	u_char	Length of Key field
HeadersLen	u_short	Length of Headers field
SessionKey	*SessionKeyLen* bytes	Session key (max 255 bytes)
Headers	*HeadersLen* bytes	Session headers (max 65535 bytes)

The ServerNonce field contains a nonce to be used as an authentication challenge to the client. Playback attacks are prevented by virtue of the fact that a nonce is a non-repeating number. Thus, every SessionReply is unique in that it will always have a different nonce.

The ClientNoncePlus field contains the authentication response from the server which is the ClientNonce from the SessionRequest incremented by one.

The ServerSessionId contains the server session identifier. The client uses this session identifier for all subsequent communication to the server.

The Cipher field contains a proposed cipher algorithm to use for the remainder of the session. If the client is capable of the proposed cipher, it may use it. Otherwise, it should use the original cipher it proposed.

The SessionKeyLen and HeadersLen fields contain the length of the SessionKey and Headers fields respectively.

The SessionKey field contains the key used to encrypt subsequent communications in the session.

The Headers field contains headers that apply to the entire session. Depending on the header key, these can be headers that will be automatically attached to subsequent service responses, or they can be session-specific parameters.

Reply

Reply is the reply PDU used to return information from the server in response to Get and Post.

TABLE 2-15. Reply Fields

Name	*Type*	*Purpose*
HeadersLen	u_short	Length of the Headers field
DataLen	u_short	Length of the Data field
Headers	*HeadersLen* bytes	Reply headers
Data	*DataLen* bytes	Reply data

The HeadersLen and DataLen fields specify the length of the Headers and Data fields respectively.

The Headers field contains the reply headers.

The Data field contains the data returned from the server.

Error

Error is a reply PDU that indicates that there was some form of error servicing the request.

In some instances, such as the server side of the session no longer existing, the error will be returned to client session 0 rather than the client session the request came from. To reduce the opportunity for denial-of-service attacks, the Error PDU includes a *tag* which is derived from the request that triggered the error. This allows the client to validate all errors coming to client session 0.

As in HTTP, each error has an error code associated with it. Also, the error can return some arbitrary data (such as an HDML deck) that further describes the error and allows the user to take corrective action.

TABLE 2-16. Error Fields

Name	*Type*	*Purpose*
Tag	4 bytes	Validator tag matching request that caused error
Code	u_short	Error code
HeadersLen	u_short	Length of the Headers field

Name	*Type*	*Purpose*
DataLen	u_short	Length of the Data field
Headers	*HeadersLen* bytes	Error headers
Data	*DataLen* bytes	Error data

The Tag field allows the client to validate that the error was actually caused by an outstanding request. If the request was sent to a user session (i.e., not server session 0), then the Tag field will contain the SessionId from the request. If the request was sent to session 0, the Tag field will contain a PDU-dependent value. In the case of SessionRequest, it will contain the four-bytes starting at the ClientNonce field.

The Code field contains a numeric code to identify the error. Error codes are listed in Table A-7 in Appendix A.

The HeadersLen and DataLen fields specify the length of the Headers and Data fields respectively.

The Headers field contains the error headers.

The Data field contains the data returned from the server.

Redirect

The *Redirect* PDU may be returned in response to a SessionRequest. It contains one or more addresses to which the SessionRequest should be sent again. This is used to migrate clients to servers whose addresses have changed, or to perform a crude form of load balancing at session creation time.

Note that this PDU will be sent to client session 0. Thus, it has the same validation tag in it that the Error PDU has.

TABLE 2-17. Redirect Fields

Name	*Type*	*Purpose*
Tag	4 bytes	Validator tag matching request that triggered redirect
AddressesLen	u_char	Length of the Addresses field
Addresses	*AddressesLen* bytes	One or more addresses to which to send subsequent SessionRequests

The Tag field contains the validator tag as described in "Error".

The AddressesLen field contains the length of the Addresses field.

The Addresses field contains one or more new addresses for the server. Subsequent SessionRequests should be sent to these addresses instead of the server address just tried. The format of the addresses is dependent on the underlying datagram protocol.

Acknowledgment PDUs

SessionComplete

SessionComplete is an acknowledgment PDU that completes the session creation process. It is sent to the server session identified in the SessionReply PDU. Thus, the entire PDU should be encrypted.

TABLE 2-18. SessionComplete Fields

Name	*Type*	*Purpose*
ServerNoncePlus	u_short	ServerNonce+1 response for authentication

The ServerNoncePlus field contains the authentication response from the client which is the ServerNonce from the SessionReply incremented by one.

Cancel

The *Cancel* PDU causes the current transaction to be canceled. The Cancel PDU has no type specific contents.

Ack

The *Ack* PDU acknowledges that the reply has been received from the server. The Ack PDU provides some information about how long the request took so that performance monitoring of the network can be performed at the server.

TABLE 2-19. Ack Fields

Name	*Type*	*Purpose*
Delay	10 bits	Round-trip time for request
Tries	6 bits	Number of retransmitted request packets

The Delay field contains the time between when the request was initiated and when a reply or error was received. It is encoded as follows:

- Values 0 to 159 are used to encode round-trip times from 0ms to 10176ms in 64 ms increments.
- Values 160 to 1022 are used to encode round-trip times from 10240ms to 892828ms in 1024ms increments.
- Value 1023 encodes a round-trip time great than 892928ms.

Some example C code to implement the delay encoding is as follows:

```
if (round_trip < 10240) {
       delay = round_trip >> 6;
}
else if (round_trip >= 10240 && round_trip <= 892928) {
       delay = ((round_trip - 10240) >> 10) + 160;
}
else {
       delay = 1023;
}
```

The Tries field contains the number of retries the client issued before a reply or error was received.

Other PDUs

HoldOn

The *HoldOn* PDU is sent from the server to the client to indicate that it has received and is processing the request. When a client receives a HoldOn, it should stop retransmitting the original request, and simply wait for the subsequent reply.

The HoldOn PDU has no type-specific contents.

Signal

The *Signal* PDU is sent to the client to indicate that data push notifications are pending, and the client should retrieve any outstanding notifications by sending the GetNotification request.

The Signal PDU contains no type-specific information. If a Signal is sent to session 0, and the client does not currently have a session with the server, it should first create one and then submit the GetNotification request.

3 Protocol Operations

This section describes how PDUs are used to perform higher level tasks such as session creation and client requests.

Protocol Stack

HDTP makes very few assumptions about the underlying protocol, namely:

- Datagrams may be sent between client and server
- Delivery of the datagrams may be unreliable and/or unordered, and it might introduce errors in the datagrams
- The addresses of client and server will not change during the lifetime of a single transaction

The assumption about datagrams does not necessarily mean that HDTP must be implemented on top of a datagram protocol. It is entirely possible to send datagrams via a connection-oriented data path.

The assumption about the addresses of client and server is actually intended to be liberating. HDTP gracefully handles the case where the underlying address of either client or server changes. HDTP sessions may span multiple network addresses over time. This allows HDTP to be implemented on devices that roam or whose address otherwise need to change with some frequency.

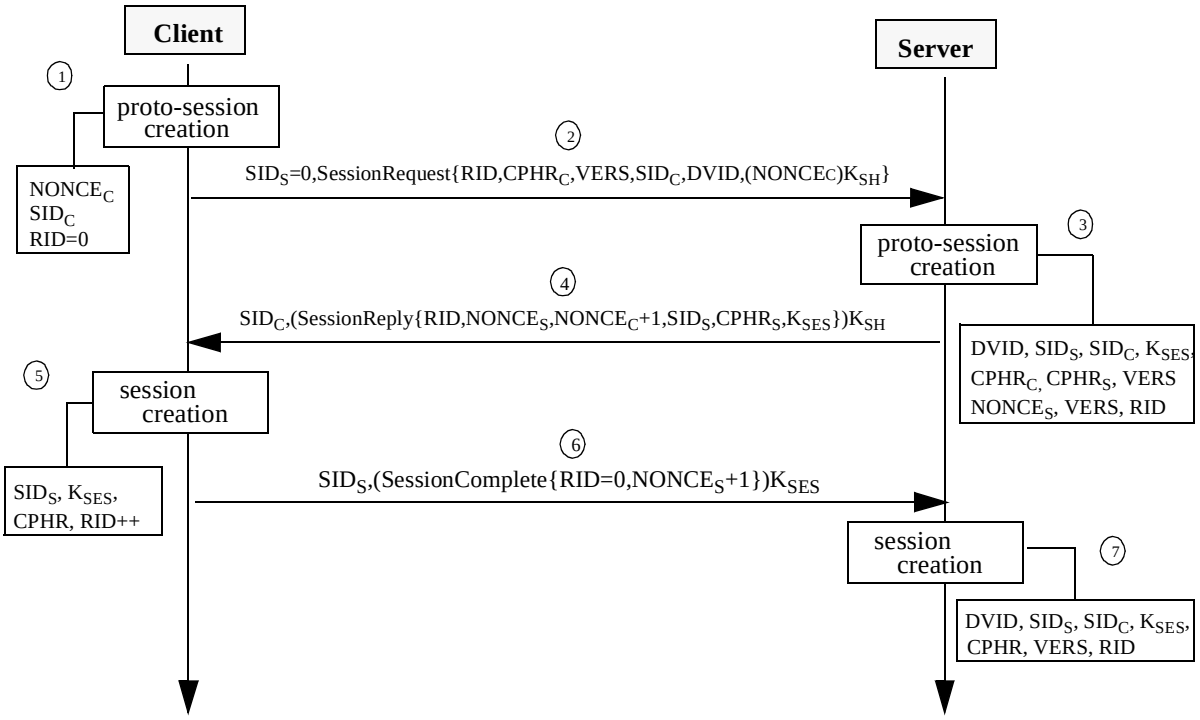
Session Management

Session management involves both creating sessions when needed, and responding to session death. HDTP sessions are not explicitly terminated. Rather, they are garbage collected by each side as necessary. The protocol supports a means to discover that one side of the session has been garbage collected and to recreate the session as necessary.

The session creation process is initiated by a client that wishes to establish an HDTP session with a server. Figure 3-1 shows the sequence of events during the session creation process between a client and server.

Session Creation

FIGURE 3-1. Session Creation Process



In the above figure, the syntax (X)K denotes that data X is checksummed and encrypted using key K. The syntax F{Y,Z} denotes the F PDU with Y and Z as its type-specific contents. The Headers fields have been left out of all the relevant PDUs for brevity. The abbreviations are defined as follows:

- SID_S, SID_C: Server and Client SessionId
- RID: Current RequestId
- $NONCE_S$, $NONCE_C$: Server and Client Nonce
- $CPHR_C$, $CPHR_S$: Client and Server Cipher Proposals
- CPHR: Final agreed upon cipher
- VERS: Version of the SUGP protocol
- DVID: DeviceId
- K_{SH}: the shared secret key between the browser and UPLink
- K_{SES}: the session key issued by UPLink

1 The client initiates session creation by first creating a client proto-session with $NONCE_C$, session identifier SID_C, and an initial RID of 0. This proto-session will decrypt packets sent to it using the shared key, K_{SH}, and the client's default cipher algorithm, $CPHR_C$

2 The client sends a *SessionRequest* to server session 0. The client sends all the information necessary to establish a session with the server, including the proposed cipher, protocol version, client session identifier, and headers. To authenticate the

server, the browser also sends the client nonce to be used for challenge-response authentication. The client nonce is encrypted with the shared secret key using the cipher algorithm proposed by the client.

3 Upon receiving the *SessionRequest*, the server looks up the shared secret key, K_{SH}, for the device identified with DeviceId. If the shared secret is found, a proto-session is created for the device. If a proto-session for that device already exists, then it is reused, otherwise, the server generates a new proto-session with a server session identifier, a session key, and a server nonce. The server also saves all information in the request for the session.

4 The server sends a *SessionReply* to the client proto-session with the server nonce, the incremented client nonce (as the response to the client challenge), the server session identifier, a proposed cipher, and the session key, and headers. This message is encrypted using the shared secret key and the cipher protocol proposed by the client in the session request.

5 At this point, the client checks the challenge response from the server to authenticate that it is really holds the shared secret key K_{SH}. If the server is authenticated, the client commits the session, saving the server session id, the session key, and deciding on which cipher to use for the session. To decide on the cipher, the client should support the cipher proposed by the server if it is stronger and if the client supports it. The client may select the cipher proposed either by the client or by the server, but no other cipher. The client also increments the current request id.

6 The client sends a *SessionComplete* to the server proto-session with the incremented server nonce (as the response to the server challenge). This message is encrypted using the session key and the proposed cipher the client decided to use.

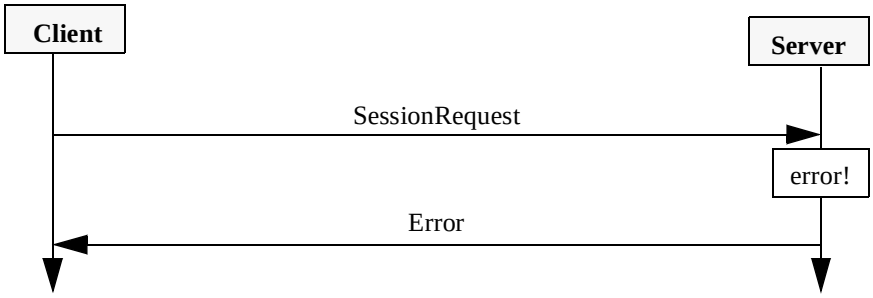
7 On receiving the *SessionComplete*, the server attempts to determine which cipher the client selected and to authenticate the client. To do this, it decrypts the message with the server's proposed cipher and checks the incremented nonce. If that fails, it then decrypts the message with the client's proposed cipher and checks the incremented nonce. If either succeeds, the server has determined which cipher to use, and that the browser indeed has the shared secret key. If the browser is authenticated, the session is created; otherwise, the corresponding proto-session is discarded. If there is a pre-existing session with the client, it is automatically terminated.

Session Errors

There are two types of session errors. Errors that occur during session creation, and an error that is caused because one side of the session or the other was garbage collected.

Errors During Session Creation

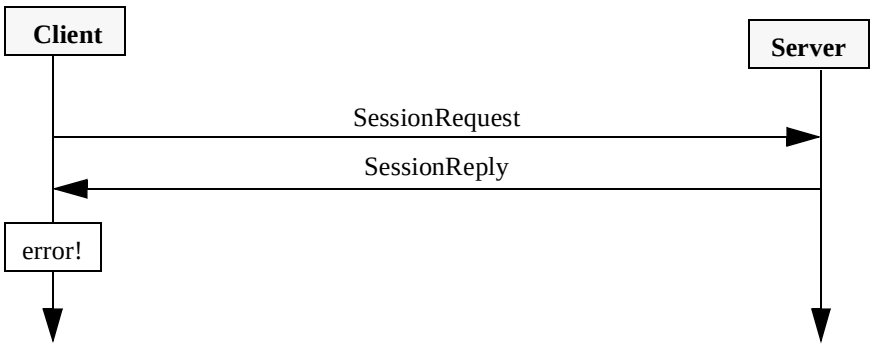
FIGURE 3-2. Session Error Detected by Server



Errors during session creation can be detected by either the server or the client. If the error is detected by the server, it sends an error packet as shown in Figure 3-2.

Note the client does not acknowledge this error. If the Error packet is lost, the client will eventually retry the SessionRequest resulting in another error. Also note that the error is sent to client session 0.

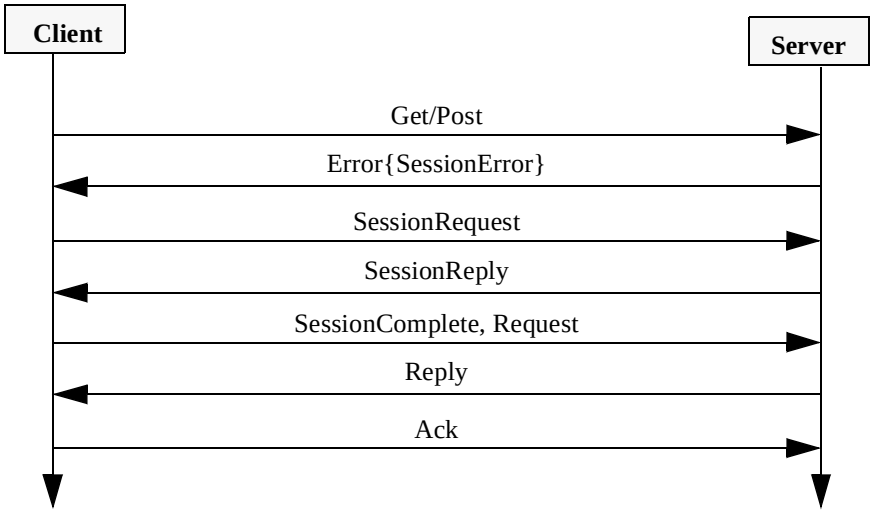
FIGURE 3-3. Session Error Detected by Client



If a session error is detected by the client, it simply terminates the session creation and stops sending SessionRequest packet as shown in Figure 3-3.

Session Error During Normal Request

FIGURE 3-4. Server Session Error During Normal Request



A session error during a normal request may occur if one side of the session or the other has been garbage collected. If the server session no longer exists, and the client attempts to send a message to it, an Error PDU containing a *SessionError* will be sent to client session 0. The client's normal response to a session error is to create a new session as shown in Figure 3-4. This example also shows the advantages of piggybacking PDUs. The SessionComplete and Request PDUs have been combined into one message, reducing the number of packets sent.

Note that the error message is sent to client session 0 which is not a secure session. The client must verify that the error tag matches the request that caused the error, and should not terminate the existing session until a valid SessionReply has been returned from the server.

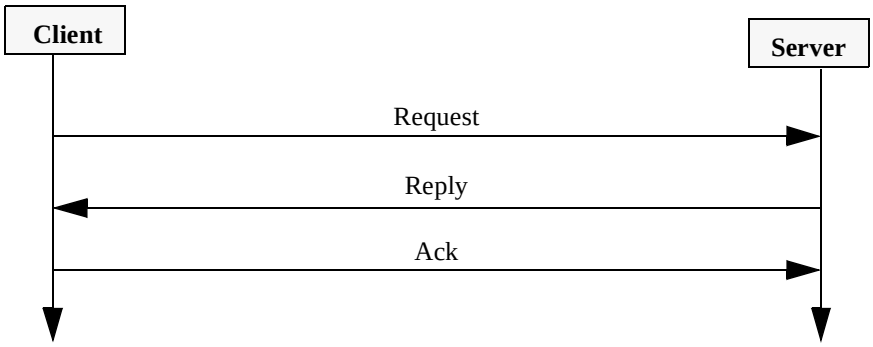
Client session errors are not reported to the server. If a message arrives at the client for a session that does not exist, it is ignored.

Normal Transactions

Once a session has been created, normal transactions can occur. In the following diagrams we use *Request* to describe any PDU from the set of request PDUs.

Three-Way Handshake

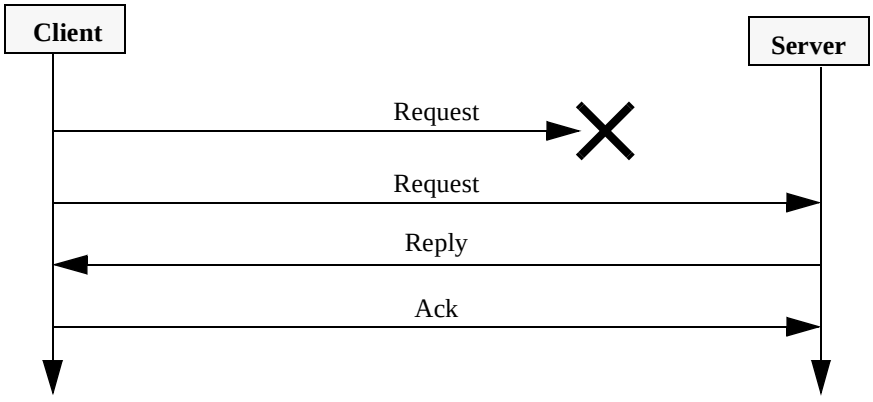
FIGURE 3-5. Normal Three-Way Handshake



The basic transaction consists of a three-way, request-reply-acknowledge handshake, as shown in Figure 3-5

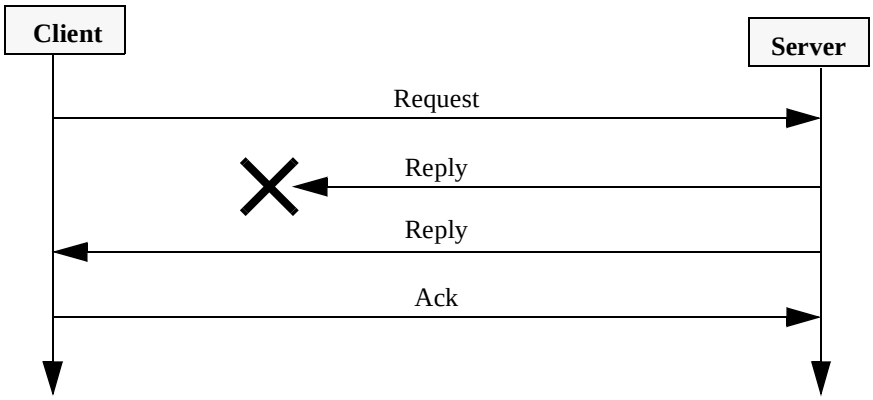
Lost Messages and Retries

FIGURE 3-6. Request Retry



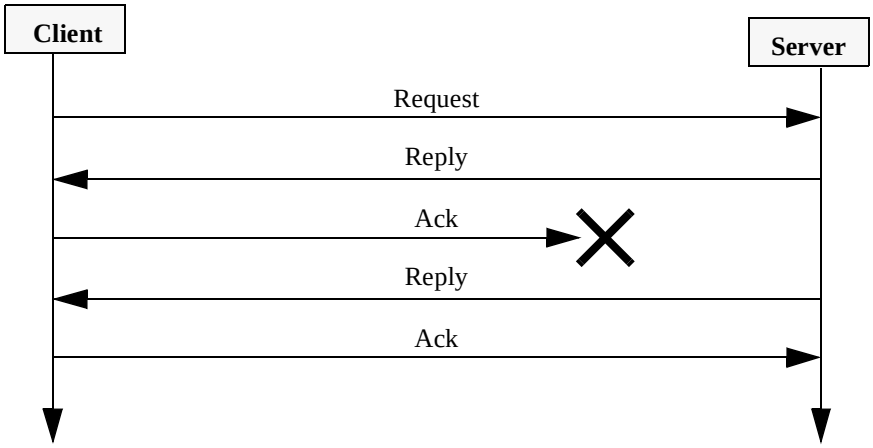
In the event of a lost request, the client continues to send the request until it receives some kind of reply as shown in Figure 3-6.

FIGURE 3-7. Reply Retry



Similarly, in the event of a lost reply, the server continues to send the reply until it receives some kind of acknowledgment as shown in Figure 3-7. If the server receives a repeated request, because the client sent another request not having received the reply, the server ignores it. The request has been processed, and any lost reply will be resent in due time.

FIGURE 3-8. Acknowledgment Retry

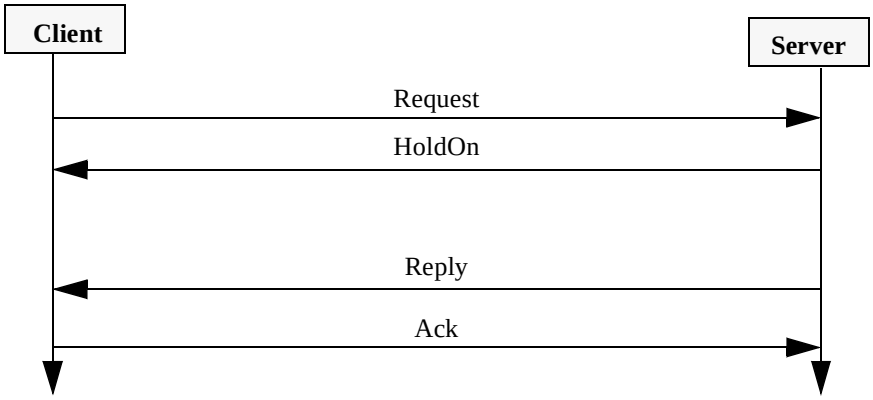


In the event of a lost acknowledgment, the server re-sends the reply, and this causes the client to re-send the acknowledgment as shown in Figure 3-8

The algorithm to calculate when and how many times to re-send a message depends on the underlying datagram protocol stack. For example, if using unreliable datagrams, an implementation should use an exponential backoff algorithm with a fixed number of retries. If using a reliable datagram or connection protocol, an implementation would not need to re-send messages at all.

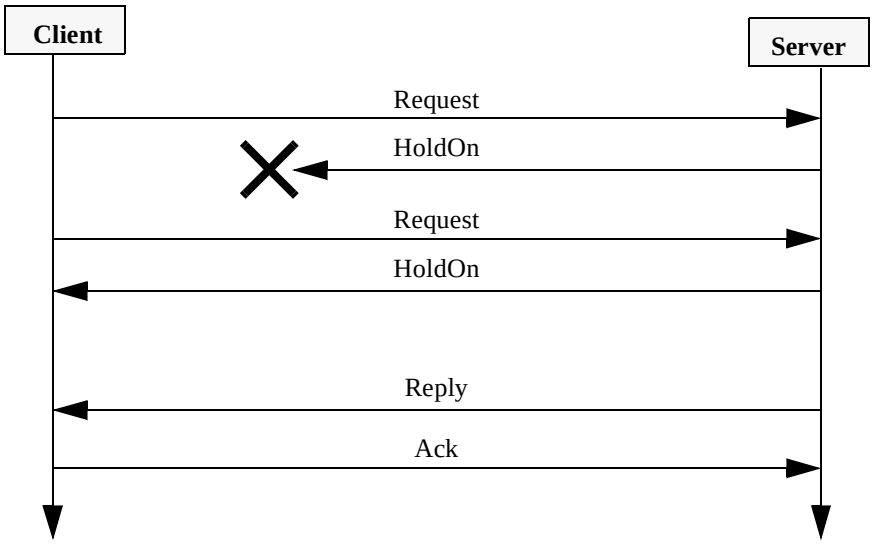
HoldOn

FIGURE 3-9. HoldOn



Often, the server will take longer to service a request than the client's retry interval. Since the client can not distinguish this from a lost request, the server will send a HoldOn PDU to the client to indicate that the client should stop sending requests.

FIGURE 3-10. HoldOn Retry



To handle the case of lost HoldOn messages, the server will send another HoldOn if it receives a repeated request. Since the client continues to re-send requests until it receives something, a HoldOn message is bound to get through eventually.

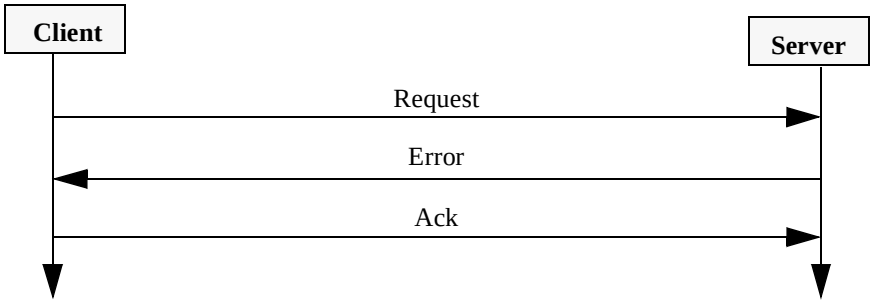
Transaction Errors

The server will return an Error PDU to the client session if it can not process the request normally. Common errors include:

- Security violation (e.g., message integrity failure)

- Service unavailable (e.g., service down)
- Unauthorized access (e.g., user/password required)
- Invalid service request (e.g., non-existent URL)

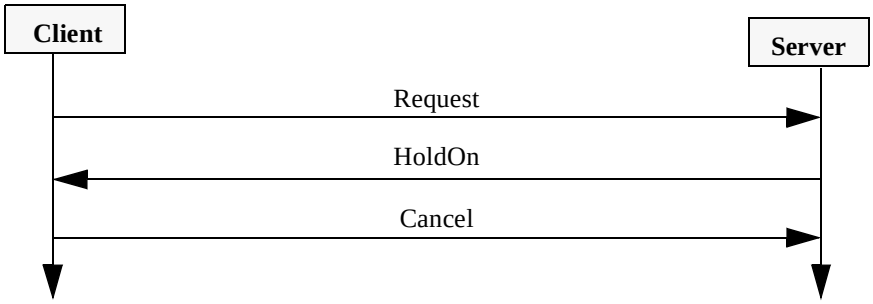
FIGURE 3-11. Error Handling



If an error is sent to a client session, the client should acknowledge receipt of the error as if it were a reply as shown in Figure 3-11.

Cancellation

FIGURE 3-12. Cancel

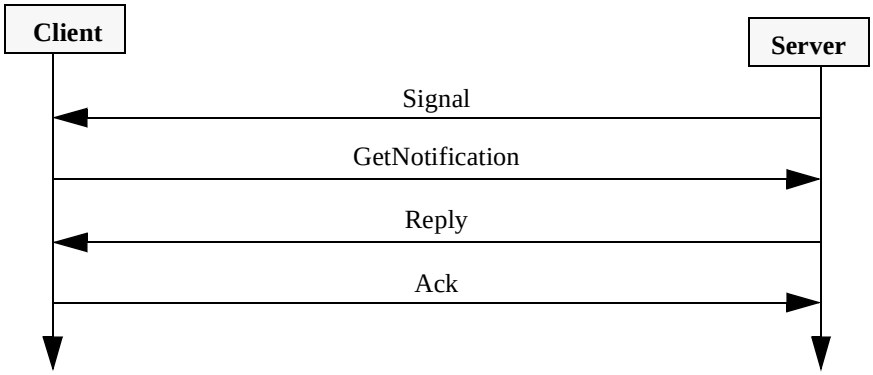


The client may cancel a transaction in progress at the user's demand by sending a Cancel PDU to the server as shown in Figure 3-12. Note that the cancel may be sent before or after a server sends the actual reply.

If the client receives a reply for a transaction that has been cancelled, it should re-send the Cancel PDU. This handles the case of the lost Cancel.

Notifications

FIGURE 3-13. Data Push Notification



The server initiates a data push notification either by sending a Signal PDU to the client or by attaching the *x-up-notify* header to the reply of a transaction in progress. The data push notification is shown in Figure 3-13.

Note that the Signal PDU is content free. In implementations where the client creates an HDTP session with only one server (such as a proxy), the Signal PDU can be sent to session 0, even through an alternate data path such as SMS. This content free signal would cause the client to send the GetNotification request to the server, potentially creating a new session or bringing up a new data connection, if needed.

If there are multiple data push notifications pending, the server should attach the x-*up-notify* header to the Reply PDU for the first notification. This will signal that another notification is pending. The client can both acknowledge the first notification and request the second notification by piggybacking an Ack PDU with a GetNotification PDU.

Assigned Numbers

This section contains tables of all the HDML assigned numbers.

TABLE A-1. Server Session Identifiers

SessionId	Purpose
0x00000000	Session creation and other meta-functions
0x00000001– 0x0000000F	Reserved
0x00000010– 0xFFFFFFFF	User sessions

TABLE A-2. Client Session Identifiers

SessionId	Purpose
0x00	Unencrypted session
0x01–0x02	SUGP backwards compatibility
0x03–0x0F	Reserved
0x10—0xFF	User sessions

TABLE A-3. PDU Types

Type	*Name*
1	SessionRequest
2	SessionReply
3	Get
4	Reply
5	GetNotification
6	Error
7	Post
8	HoldOn
9	Cancel
10	Signal

Type	*Name*
11	Ack
12	*reserved*
13	SessionComplete
14	Redirect
15	*reserved*
16	Options
17	Head
18	Put
19	Delete
20-127	*unassigned*

TABLE A-4. Cipher Algorithms

Cipher	*Description*
0	No encryption; second byte undefined.
1	RSA RC5 32 bit words, 16 byte keys, Block Padding, No IV, variable rounds specified in second byte; 16 rounds minimum.

TABLE A-5. Well-known Header Key Encodings

Key	*Encoding*
Location	0x80
x-up-time	0x81
x-up-notify	0x82
x-up-retry	0x83
x-up-errlmt	0x84
x-up-maxpdu	0x85
x-up-disp	0x86
home	0x87
Accept-Charset	0x88
Accept-Language	0x89
x-up-cap	0x8A
x-up-devtype	0x8B
Accept	0x8C
Accept-Encoding	0x8D
Accept-Ranges	0x8E
Age	0x8F
Allow	0x90
Authorization	0x91

Key	*Encoding*
Cache-Control	0x92
Connection	0x93
Content-Base	0x94
Content-Encoding	0x95
Content-Language	0x96
Content-Location	0x97
Content-MD5	0x98
Content-Range	0x99
Content-Type	0x9A
Date	0x9B
ETag	0x9C
Expires	0x9D
From	0x9E
Host	0x9F
If-Modified-Since	0xA0
If-Match	0xA1
If-None-Match	0xA2
If-Range	0xA3
If-Unmodified-Since	0xA4
Last-Modified	0xA5
Location	0xA6
Max-Forwards	0xA7
Pragma	0xA8
Proxy-Authenticate	0xA9
Proxy-Authorization	0xAA
Public	0xAB
Range	0xAC
Referer	0xAD
Retry-After	0xAE
Server	0xAF
Transfer-Encoding	0xB0
Upgrade	0xB1
User-Agent	0xB2
Vary	0xB3
Via	0xB4
Warning	0xB5
WWW-Authenticate	0xB6

TABLE A-6. Well-known Content-Type Header Encodings

Content-Type	*Encoding*
application/x-up-digest	0x80
application/x-hdmlc	0x81

TABLE A-7. Error Codes

Code	*Error*
1	Device Error, unrecognized DeviceId
2	Key Error, invalid shared secret key
3	Session Error, invalid session
4	Transaction Error, request failed

Bibliography

B

[HDML]

Unwired Planet, Inc., "Handheld Device Markup Language Specification", April 1997, http://www.uplanet.com/pub/hdml_w3c/hdml20-1.html.

[RFC822]

Crocker, David H., "STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES", RFC 822, University of Delaware, August 1982, http://www.internic.net/rfc/rfc822.txt.

[RFC1521]

N. Borenstein, et. al., "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies", RFC 1521, Bellcore, September 1993, http://www.internic.net/rfc/rfc1521.txt.

[RFC1630]

Berners-Lee, T., "Universal Resource Identifiers in WWW, A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web", RFC 1630, CERN, June 1994, http://www.internic.net/rfc/rfc1630.txt.

[RFC1737]

Sollins, K., and L. Masinter, "Functional Requirements for Uniform Resource Names", RFC 1737, MIT/LCS, Xerox Corporation, December 1994, http://www.internic.net/rfc/rfc1737.txt.

[RFC1738]

Berners-Lee, T., et. al., “Uniform Resource Locators (URL)”, RFC 1738, CERN, December 1994, http://www.internic.net/rfc/rfc1738.txt.

[RFC1808]

Fielding, R., “Relative Uniform Resource Locators”, RFC 1808, UC Irvine, June 1995, http://www.internic.net/rfc/rfc1808.txt.

[RFC2047]

Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text", RFC 2047, University of Tennessee, November 1996, http://www.internic.net/rfc/rfc2047.txt.

B Bibliography

[RFC2068]

Fielding, R., et. al., “Hypertext Transfer Protocol -- HTTP/1.1”, RFC 2068, UC Irvine, January 1997, http://www.internic.net/rfc/rfc2068.txt.

[SSL]

Freier, Alan O., et. al., "The SSL Protocol, Version 3.0", Internet Draft, Netscape, March 1996, http://home.netscape.com/eng/ssl3

[STD1]

Postel, J., Editor, "INTERNET OFFICIAL PROTOCOL STANDARDS", STD 0001, Internet Architecture Board, February 1997, http://www.internic.net/std/std1.txt.

[STD2]

Reynolds, J., Postel, J., "Assigned Numbers", STD 0002, ISI, October 1994, http://www.internic.net/std/std2.txt.

[STD3]

Braden, R., "Requirements for Internet hosts - application and support", STD 3, RFC 1122, IETF, October 1989, http://www.internic.net/std/std3.txt.

[STD5]

Postel, J., "Internet Protocol", STD 0005, RFC 791, Information Sciences Institute, September 1981, http://www.internic.net/std/std5.txt.

[STD6]

Postel, J., "User Datagram Protocol", STD 0006, RFC 768, ISI, August 1980, http://www.internic.net/std/std6.txt.

[STD11]

Crocker, David H., "STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES", STD 0011, RFC 822, University of Delaware, August 1982, http://www.internic.net/rfc/std11.txt.